



Carcassonne

PROJET S6

Hadhami Ben Abdeljelil, Arthur Jolivet, Marie Ostertag,
Baptiste Roseau

ENSEIRB-MATMECA

18 mai 2018

Encadrant : Steve NGUYEN

Table des matières

1	Présentation du sujet	2
1.1	Règles	2
1.2	Contraintes du sujet	3
2	Environnement de travail	4
2.1	Compilation et exécution	4
2.2	Environnement de Développement Intégré	4
3	Conception de la solution algorithmique	5
3.1	Architecture du jeu	5
3.2	Types abstraits de données	6
3.2.1	Pile	6
3.2.2	File	6
3.2.3	Set	7
3.3	Présentation des structures de données	8
3.3.1	Structure de données board	8
3.3.2	Structure de données card	9
3.3.3	Structure de données player	10
3.3.4	Structure de données pour la gestion des zones et du score	11
4	Correction et complexité des algorithmes	14
4.1	Vérification de correspondance et ajout d'une carte	14
4.2	Calcul des coups possibles	15
5	Validation et qualité du développement	16
5.1	Tests unitaires	16
5.2	Vérification par mise en commun des clients	17
5.3	Interface utilisateur	18
6	Problèmes rencontrés	19
6.1	Vérification de correspondance entre 2 cartes	19
6.2	Utilisation de fichiers JPEG pour l'affichage des cartes avec SDL2	19
6.3	Ajout de copies dans les TADs	20
6.4	Interprétation différente du sujet selon les groupes	20
7	Pistes d'amélioration	21
7.1	Fonctionnalités restantes : placement des <i>meeples</i>	21
7.2	Fonctionnalités restantes : comptage des points	21
7.3	Fonctionnalités restantes : stratégies de jeu	21
7.4	Conception des TADs sans copie	22
8	Conclusion	23

1 Présentation du sujet

Cette section présente le projet C de sixième semestre qui consiste à implémenter le jeu de plateau *Carcassonne*, et les contraintes qui lui sont liées.

1.1 Règles

L'objectif de ce projet est d'implémenter le jeu Carcassonne en respectant une architecture client serveur afin d'y faire jouer des joueurs de différents groupes de projet en nombre quelconque. Cette première partie permet de rappeler les règles de ce jeu de société.

Cartes

La boîte de jeu est constituée de 72 cartes de paysage de 23 sortes différentes, représentant des types de zones différentes : chaque carte peut être découpée en neuf zones représentant soit un chemin, soit un carrefour, soit une abbaye, soit un quartier de ville, soit un pré.

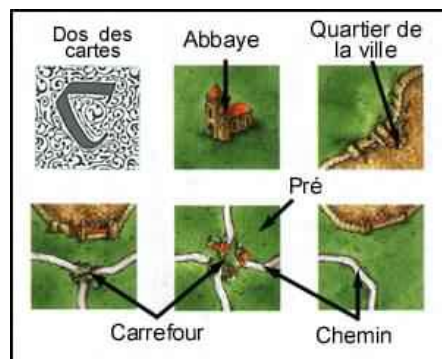


FIGURE 1 – Visuel de certaines cartes (tiré du site jeuxstrategie.free.fr dont le lien est fourni dans le sujet

Déroulement

L'objectif de chaque joueur est de marquer un maximum de points en positionnant sur les cartes ses partisans qui sont des petites figurines en bois. Le vainqueur est celui qui a le plus de points à la fin de la partie. En tout début de partie, la carte de départ (toujours la même) est placée sur la table.

Une pile des cartes est positionnée sur la table, les joueurs jouent les uns après les autres jusqu'à ce qu'il n'y ait plus de carte dans la pioche. Quand un joueur a pioché, il doit poser la carte piochée à une position autorisée sur le plateau, c'est-à-dire que les bordures de cette carte doivent coïncider avec celles des cartes adjacentes. Puis il a le droit de mettre un de ses partisans sur un chemin, dans

une ville, un pré ou une abbaye sur la carte qu'il vient de positionner.

Comptage des points

Si à la fin de son tour un joueur a un de ses partisans dans une ville achevée (i.e clôturée par des remparts et sans espace vide), ou sur un chemin clôturé, il récupère son partisan et marque les points associés à cette zone, à savoir 2 points par carte où se trouve la ville ou le chemin, plus les éventuels blasons à raison de deux points par blason.

Enfin, il y a un comptage des points à la fin de la partie, chaque joueur marque 1 point par carte où il a possédé un chemin ou une abbaye, plus éventuellement 2 points par carte avec blason. Et pour chaque pré, le ou les joueurs qui possèdent le plus de partisans sont déclarés propriétaires du champ et marquent un point par carte où s'étend le champ.

Pour de plus amples renseignements à propos des règles du jeu *Carcassonne*, se référer à [la règle du jeu](#).

1.2 Contraintes du sujet

Le langage C est imposé pour implémenter le projet, ce qui implique notamment la gestion de l'allocation dynamique de la mémoire. Néanmoins, ce langage offre la possibilité de réutiliser nos connaissances autour des types abstraits de données.

L'utilisation du système de contrôle de version *Subversion*, hébergé sur le serveur de l'école, ainsi que de l'outil de compilation *make* était également imposé.

Une particularité est de décomposer le projet en deux parties distinctes : les clients et le serveur de jeu. L'intérêt de cette représentation est de créer des clients qu'il est possible d'interchanger. Chaque client suit une stratégie de jeu qui lui est propre. Ainsi, la première étape consiste à transformer chaque fichier client en une bibliothèque dynamique, qui sera passée en paramètre à un serveur de jeu.

2 Environnement de travail

2.1 Compilation et exécution

Afin de travailler sur le projet nous avons utilisé, comme indiqué en contrainte l'outil de compilation *make* utilisant un **Makefile**, mais nous avons également utilisé l'outil *Cmake* nécessitant l'écriture d'un fichier **CMakeLists.txt** car l'utilisation de celui-ci était nécessaire pour travailler avec un environnement de développement intégré (ou **IDE**¹).

Concernant le *Makefile*, nous avons incorporé plusieurs règles afin d'automatiser toutes commandes dont nous avons besoin. Nous avons donc les règles suivantes :

- **build**, permettant de compiler le serveur et les clients.
- **install**, installant le server et les clients dans le dossier *install/* et supprimant les fichiers objets.
- **run** et ses variantes **vrun** et **grun**, exécutant le programme respectivement en mode console, en mode console avec analyse de la mémoire, en mode graphique.
- **test** et ses variantes **vtest** et **ctest** exécutant les tests respectivement en affichage complet, en affichage résumé avec indications sur la mémoire, en affichage résumé avec la couverture des tests.
- **docs** générant la documentation à l'aide de *Doxygen*².
- **clean**, permettant de supprimer tous les fichiers générés par les précédentes commandes (dont la documentation).

2.2 Environnement de Développement Intégré

Comme indiqué au paragraphe précédent, nous avons utilisé un **IDE** afin de travailler de manière plus efficace.

En effet, celui-ci centralise toutes les étapes de réalisation sur un seul logiciel jouant les rôles, entre autres, d'éditeur de code avec auto-complétion et inspection en temps réel, de compilateur, de débogueur, d'analyseur de mémoire, de contrôleur de version, de moteur de recherche et remplacement sur le projet complet pour un refactoring de code instantané.

L'utilisation de **CMake** qui est un outil générique de compilation, nous a permis de choisir des IDE différents suivant notre convenance ainsi que les contraintes du matériel utilisé. Ainsi nous avons utilisé 3 IDE différents : QT Creator, CLion ou encore Visual Studio Code, ce dernier étant cependant plus un éditeur de code qu'un IDE.

1. Integrated Development Environment, Environnement de Développement Intégré (EDI) en français. C'est un éditeur de texte équipé de fonctionnalités permettant aux développeurs de gagner en efficacité dans la production de leur code.

2. Logiciel libre générateur de documentation. Doxygen et GraphViz (Dots) doivent être installés et accessible dans le *Path*

3 Conception de la solution algorithmique

Cette section s'intéresse à la conception du projet Carcassonne à travers les types abstraits de données que nous avons choisi d'implémenter pour représenter le jeu.

3.1 Architecture du jeu

Boucle de jeu

Les règles du jeu nous ont amenés à nous intéresser aux différents points suivants : comment représenter une carte ? Comment représenter le plateau ? Comment représenter un emplacement de *meeple* ? Comment compter les points ?

Le pseudo-algorithme global que nous allons suivre est le suivant ³ :

```
moves = empty_list()
For each player p
  p->initialize(p_id, num_players)
While (the drawing pile is not empty)
  c = draw_card()
  p = compute_next_player()
  m = p->play(c, last_n(moves))
  enqueue(m, moves)
For each player p
  p->finalize()
```

Afin d'obtenir un code aussi simple, plusieurs TADs sont nécessaires. Nous avons choisi d'implémenter une pile pour la pioche, une file pour stocker les coups et les joueurs, et enfin un ensemble (nommé *set*) permettant de stocker de façon ordonnée les cartes sur le plateau de jeu. Ce dernier sera celui développé en PG116 pour éviter d'avoir à repartir de zéro.

De façon à pouvoir être utilisés à des endroits différents, les TADs que nous avons utilisés sont polymorphes, c'est à dire qu'ils stockent des *void** (pointeurs génériques).

Nous utilisons un tableau de taille variable alloué sur le tas, en multipliant sa capacité par 2 s'il y a dépassement, afin de ne pas être limité. À l'initialisation, l'utilisateur devra fournir des fonctions de copie, de suppression et d'affichage adaptées aux objets qu'il souhaite manipuler. Pour le TAD *set*, il devra aussi fournir une fonction de comparaison.

3. Tiré du [sujet du projet](#).

Raisons de ces choix

La pioche a juste besoin d'être parcourue une seule fois, sans nécessité de remettre en jeu une carte, même si la carte piochée n'est pas jouable (auquel cas elle est perdue). Ainsi la pile est une bonne solution, qui sert de point d'arrêt à la boucle principale, quand celle-ci est vide.

Les joueurs jouent tour à tour, en piochant une carte au début de leur tour, et la partie s'arrête quand la pioche est vide. Pour actualiser leur vision du jeu, ils ont besoin de savoir ce qu'on joué tous les joueurs précédents durant ce tour, c'est donc plus logique de les représenter par une file.

Enfin, une structure ordonnée a été implémentée pour le stockage des cartes, afin d'y accéder plus facilement et plus rapidement, et cela permet également d'avoir accès à toutes les caractéristiques de celle-ci. Mais la structure *set* est aussi utilisée pour gérer les zones du plateau de jeu, ce qui permettrait entre autres de compter les points en cours de partie si cette fonctionnalité était terminée.

3.2 Types abstraits de données

3.2.1 Pile

La pile contient un tableau *array*, sa capacité *capacity* et un indice de tête de pile *head*.

```
1 struct stack
2 {
3     void **array;
4     size_t capacity;
5     size_t head;
6     void* (*operator_copy) (void*);
7     void (*operator_delete) (void*);
8     void (*operator_debug) (void*);
9 };
```

À chaque empilement, on copie l'élément à empiler, puis on le place sur la case de *array* correspondant à *head*, que l'on incrémente par la suite.

Pour ce qui est du dépilement, on crée une copie de l'élément situé dans *array* à l'indice *head*−1, puis on supprime l'élément dans *array* avant de décrémenter *head* et renvoyer la copie.



3.2.2 File

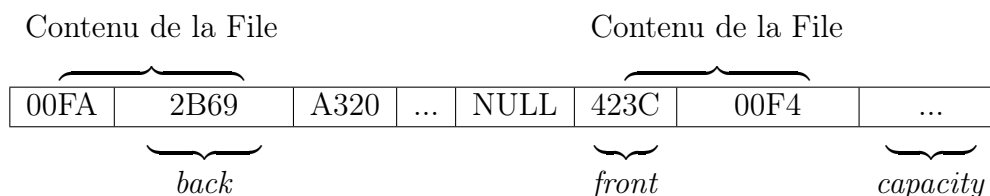
La file contient un tableau *array*, sa capacité *capacity*, un indice de tête de file *front* et un indice de fin de file *back*.

```
1 struct queue
2 {
3     void **array;
4     size_t capacity;
5     size_t front;
6     size_t back;
7     void* (*operator_copy) (void*);
8     void (*operator_delete)(void*);
9     void (*operator_debug)(void*);
10};
```

À chaque enfillement, on copie l'élément à enfiler, puis on le place sur la case de *array* correspondant à *back*. On incrémente ensuite *back capacity* de façon à optimiser l'espace disponible dans *array*.

Pour ce qui est du défilement, on peut accéder à l'élément en tête de file via la fonction `queue_front()` qui renvoie une copie de l'élément situé dans `array` à l'indice `front`.

Pour défiler, on supprime l'élément dans *array* à l'indice *front* avant d'incrémenter *back* modulo *capacity*.



3.2.3 Set

Le *set* contient un tableau trié *array*, sa capacité *capacity* et sa taille *size*. Pour le tri du tableau, l'utilisateur doit fournir une fonction de comparaison renvoyant :

- $+1$ pour $>$
- 0 pour $=$
- -1 pour $<$

```
1 struct set
2 {
3     void **array;
4     size_t capacity;
5     size_t size;
6     void* (*copy) (void*);
7     int (*cmp) (void*, void*);
8     void (*delete) (void*);
9     void (*debug) (void*);
10};
```

Pour conserver le tri, on ajoute un élément là où on le recherche. Plus précisément, à chaque ajout, on recherche l'élément à ajouter dans le tableau via une recherche dichotomique, puis, si il n'est pas déjà présent, on en place une copie dans s de façon à conserver le tri.

Pour ce qui est du retrait, on recherche l'élément du tableau, puis on le supprime

si on le trouve.

Chaque ajout ou retrait se fait en déplaçant les éléments à sa droite d'un indice vers la droite pour un ajout, et d'un indice vers la gauche pour un retrait.

00FA	2B69	...	A320	423C	NULL	00F4	...
Contenu du set (trié)			<i>size</i>		<i>capacity</i>		

3.3 Présentation des structures de données

Cette section présente l'ensemble des structures de données utilisées dans le programme.

Un certain nombre de structures de données a été implémenté :

- La structure *board* représentant le plateau de jeu. Elle est utilisé par le serveur et par les clients qui ont leurs propres visions du jeu.
- La structure *card* permettant de décrire une tuile en la découpant en plusieurs sections et de définir son orientation.
- La structure *player* représentant un joueur.
- Les structures *zone* et *partition* sont utilisées pour gérer les zones sur le plateau, ainsi que pour le comptage des points.

3.3.1 Structure de données board

Le plateau de jeu est représenté par la structure suivante :

```

1 struct board
2 {
3     struct stack* drawing_stack;
4     struct card* first_card;
5     struct set *cards_set;
6     struct queue* moves_queue;
7 };

```

Le champ *drawing_stack* stocke la pile des cartes à piocher, *first_card* stocke la première carte déposée sur le plateau (qui est imposée par le jeu Carcassonne), *cards_set* stocke l'ensemble des cartes du plateau et *meeples_set* et *moves_queue* stocke l'ensemble des coups joués lors de la partie.

Un certain nombre de fonctions relatives à cette structure de données ont été implémentées :

- *board_init()* permet la création du plateau de jeu.
- *board__init_deck_and_first_card()* prend en paramètre le plateau et initialise sa pile de cartes selon les règles de Carcassonne.
- *board__add_custom_first_card()* prend en paramètre le plateau, l'identifiant de la carte à utiliser en tant que première carte, sa position et sa direction initiale et positionne cette carte sur le plateau de jeu.

- `board__is_valid_card()` prend en paramètres un plateau de jeu et un identifiant de carte (`card_id`) et vérifie si cette carte peut être positionnée sur le plateau.
- `board__retrieve_card_by_position()`, qui récupère une carte du *set* à partir de sa position sur le plateau de jeu.
- `board__add_card()` ajoute une carte à un plateau de jeu après vérification.
- `board__free()` libère la mémoire allouée à un plateau de jeu.

Les fonctions, `board__add_meeple()`, `board__check_sub_completion()` n'ont pas pu être implémentées. Celles-ci correspondaient respectivement à l'ajout d'un *meeple* après vérification et au comptage des points.

3.3.2 Structure de données card

Une carte est représentée par la structure suivante :

```
1 struct card
2 {
3     struct card_type type;
4     struct card * neighbors[NB_DIRECTIONS];
5     enum direction direction;
6     struct position pos;
7 };
```

Avec struct `card_type` la structure :

```
1 struct card_type
2 {
3     enum card_id id;
4     enum area_type areas[MAX_ZONES];
5 };
```

Le champ *areas* est un tableau de taille 13 (correspondant au découpage en 13 zones de chaque carte) de type `enum area_type` et indexé par *enum place*. Un `enum area_type` peut voir les valeurs suivantes : *FIELD*, *CITY*, *ROAD*, *ABBEY*, ou *INTERSECTION*.

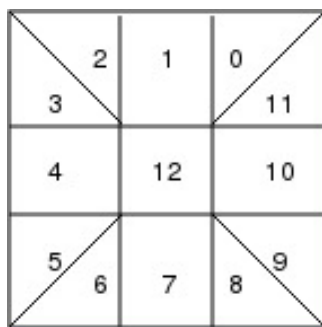


FIGURE 2 – Numérotation des *area_type* du champ *areas* d'un *card_type*

- Le champ *neighbors* de struct *card* est un tableau contenant des pointeurs vers quatre cartes entourant la carte en question.
- *direction* détermine l'orientation de la carte par rapport à *card_type* (même orientation, rotation de 90°, rotation de 180° ou rotation de 270°).
- *pos* correspond à la position de la carte sur le plateau de jeu (entier relatif qui pour la carte de départ est {0, 0}).

Un certain nombre de fonctions relatives à cette structure de données ont été implémentées :

- *card__init()* permet, à partir d'une enum *card_id* de fournir un pointeur vers la structure *card* correspondante.
- *card__free()* libère la mémoire réservée à la structure *card*.
- *card__get_relative_area()* fournit, à partir d'une enum *place* et d'une carte l'enum *area* type de cette zone en considérant la rotation.
- *card__get_neighbour_number()* fournissant le nombre de voisins d'une carte.
- *card__get_position_at_direction()* prenant une carte et une direction et renvoyant la position de sa voisine à cette direction.
- *card__are_matching_free_direction()* décide si deux cartes coïncident sur au moins une direction.
- *card__are_matching_directions()* prenant en paramètre deux cartes et deux directions et décide si les cartes coïncident suivant ces deux directions.
- *card__link_at_directions()* permet de relier deux cartes entre elles selon les deux directions fournies en argument.
- *card__unlink_neighbours()* déconnecte une carte de tous ses voisins.

Des opérateurs sont également été implémentés. Ils sont nécessaires lors de la construction de file de joueurs dans le serveur.

3.3.3 Structure de données player

Un joueur est représenté par la structure suivante :

```
1 struct player
2 {
3     unsigned int score;
4     unsigned int nb_meeples;
5     unsigned int id;
6     void* lib_ptr;
7     const char * (*get_player_name)(void);
8     const char * (*initialize)(unsigned int, unsigned int);
9     struct move (*play)(enum card_id, const struct move*, size_t);
10    void (*finalize)(void);
11 };
```

- *score* correspond au nombre de points qu'il a accumulé.
- *nb_meeples* est le nombre de *meeples* qu'il a positionné.

- *id* correspond à son numéro.
- *lib_ptr* est le pointeur vers sa librairie.
- Les autres champs correspondent aux pointeurs vers les fonctions permettant l'échange entre le serveur et le joueur.

Les fonctions relatives à cette structure de données sont les suivantes :

- *player__init()* prend un *id* et un pointeur vers une librairie et initialise un joueur.
- *player__print_name_and_score()* qui prend en paramètre un joueur et affiche sur la sortie standard son id, son nom et son score de façon ordonnée.
- *player__free()* libère la mémoire allouée à un joueur.

Des opérateurs sont également été implémentés. Ils sont nécessaires lors de la construction de file de joueurs dans le serveur.

3.3.4 Structure de données pour la gestion des zones et du score

Nouvelles structures

Cette partie mentionne des structures et fonctions qui étaient en cours de développement à la date de rendu.

Cette partie s'intéresse à la gestion des zones dans l'objectif de déterminer si un *meeples* peut être placé sur une carte dans une zone précise et afin de calculer le score des joueurs lors d'une fermeture de zone.

On appelle "zone" un ensemble de parties de cartes qui appartiennent à la même zone géographique en construction ou achevée, c'est-à-dire qui sont de même type et contiguës . Il peut s'agir d'un chemin ou d'une ville par exemple. Pour représenter les zones et les *meeples* éventuellement positionnés ceux-ci, une structure de zone a été créée :

```
1 struct zone
2 {
3     struct set *area_set;
4     struct set *meeples_set;
5     enum area_type area;
6     unsigned int score;
7     unsigned int holes;
8 };
```

Chaque zone est munie de différents champs :

- un *struct set* où l'on stocke les pointeurs vers les **area_type** qui appartiennent à la zone ;
- un *struct set* qui stocke les pointeurs vers les *struct meeples* qui sont placées sur une partie de carte comprise dans la zone ;
- *area* est le type de zone parmi *FIELD*, *CITY*, *ROAD*, *ABBEY* et *INTERSECTION* ;
- le score provisoire rapporté par l'ensemble des cartes qui forment la zone ;

- *holes* est le nombre d'emplacements libres pour placer une carte qui complète la zone.

Et toutes ces zones sont stockées dans un *struct partition* comme suit, reprenant la structure de données *struct set*.

```
1 struct partition
2 {
3     struct set * zones[MAX_ZONE];
4     int size;
5 };
```

L'idée est qu'à chaque ajout de carte sur le plateau, une ou plusieurs zones sont élargies. Pour cela, après avoir choisi l'emplacement où jouer la carte piochée, on met à jour les zones impactées. Les *area_type* de la nouvelle carte sont ajoutés aux zones du *set* correspondant si ils viennent s'intégrer dans une zone déjà existante d'une carte voisine, ou créent de nouvelles zones sinon. Dans le cas où deux zones sont reliées par une même carte, on procède à leur fusion.

On met ensuite à jour le dernier champ *holes* comme suit : à chaque ajout d'une carte, on décrémente *holes* du nombre de trous qu'elle bouche dans cette zone, et on incrémente *holes* du nombre de trous (non bouchés) ajoutés par cette carte. Ceci donne un critère pour savoir si un chemin, une ville, une abbaye ou un champs est achevé : lorsque ce compteur est à 0 il n'y a plus de place disponible dans la zone, elle est donc achevée.

Fonctions de mises à jour

Un certain nombre de fonctions relatives à l'actualisation des zones ont été implémentées :

- *zone__empty()* crée une nouvelle zone selon sa nature (*FIELD*, *CITY*, ...)
- *zone__cards_available_zones()* retourne un *struct set* de pointeurs vers les *enum place* d'une carte passé en paramètre où on peut potentiellement poser un *meeple* : cela fait référence au tableau *allowed_positions* donnée dans le fichier des prototypes de l'interface commune ;
- *zone__matching_area_set()* renvoie la zone contenant l'*area* considérée ;
- *zone__searched_allowed_positions()* prend en paramètre un pointeur vers une carte et un pointeur vers un *enum_area* lié à cette carte et retourne le *enum_area* de la même zone géographique sur la carte où on peut poser un *meeple* ;
- *zone__zones_by_card_id()* permet à partir du type de carte considéré de créer les ensembles d'*area_types* appartenant aux mêmes zones ;
- *zone__add_areas()* est la fonction qui sert à ajouter une carte aux zones qu'il faut, et réactualiser pour les zones concernées le compteur de points et *holes* indiquant le nombre d'emplacements libres des zones ;
- *zone__is_fusion_required()* renvoie un booléen s'il faut fusionner ou non deux zones initialement disjointes partant d'un seul paramètre : la carte insérée au plateau ;

- `zone__fusion_zone()` permet le cas échéant de fusionner deux zones ;
- `zone__add_meeple()` ajoute le *meeple* pointé en entrée à la zone choisie en amont ;
- `zone__free()` libère la mémoire allouée à une zone ;

Comptage des points

Pour stocker le score de chaque joueur en cours de partie on utilise une variable de type entier non signé appartenant à la structure *player*.

Comme présenté dans cette section, une variable de score est également présente dans chaque zone. À chaque fin de tour d'un joueur, il faut vérifier si une ou plusieurs zones ont été achevées. C'est le compteur *holes* présent dans chaque zone qui va permettre de les identifier. Une zone n'a pu être fermée que parce que la nouvelle carte a fermé une zone. Il suffit donc de vérifier si le compteur de chacune des zones auxquelles appartiennent des parties de la nouvelle carte ajoutée est passé à zéro ou pas.

Il y a au plus 9 zones à vérifier, 9 étant le nombre maximal d'emplacements possibles sur une même carte où on peut placer un *meeple*. Si une zone a été achevée par l'ajout de la dernière carte, en parcourant le set de *meeples* de la zone en question, il est possible de déterminer l'identité du joueur qui a le plus de *meeples* sur la zone et donc de lui attribuer les points de la zone.

Ensuite, il faut redistribuer l'ensemble des *meeples* qui étaient sur la zone, qu'ils aient permis ou non de marquer des points à son propriétaire.

Enfin, la dernière étape consiste à supprimer la zone de l'ensemble des zones en jeu, c'est-à-dire de la supprimer de la *struct partition*.

Limites de cette représentation

Si le choix de représentation simplifie les parcours de graphe (qui sont en fait remplacés par des accès au *set* stockant l'ensemble des zones), il pose aussi de nouvelles problématiques. En effet, une fois une ville achevée, elle est supprimée du *set* des zones, et ne sera donc plus accessible si on ne la stocke pas ailleurs. Cela s'avère problématique dans l'optique de calculer les points liés aux champs : il faudrait relocaliser les villes voisines des champs pour permettre le calcul des points des joueurs ayant des *meeples* sur ces champs.

4 Correction et complexité des algorithmes

4.1 Vérification de correspondance et ajout d'une carte

La fonction `board__add_card()` est responsable de la vérification de correspondance entre les cartes dans le but de les ajouter si tel est le cas. Son prototype en C est le suivant :

```
1 int board__add_card(struct board *b, struct card* card_to_add);
```

Elle se compose des fonctions énoncées ci-dessous afin de vérifier la correspondance des cartes, ajouter une carte au plateau ou encore chercher une position de voisin.

`card__are_matching_directions()` est la fonction permettant de décider de la correspondance entre deux cartes. Elle est de complexité **constante**.

`board__retrieve_card_by_position()`, qui récupère une carte du *set* à partir de sa position sur le plateau de jeu est de complexité $\theta(\log(n))$ avec n le nombre de cartes dans le plateau.

La fonction `set__add()` permettant d'ajouter un élément à un *set* est de complexité $\theta(n)$.

La fonction `board__add_card()` est écrite ci-dessous en pseudo-code avec la complexité de ses sous-fonctions. Vous pouvez trouver son code complet dans le fichier `src > server > board.c`.

```
1 check_if_position_is_free(cards_set, card) //O(log(n))
2 card = add_card_to_set(cards_set, card) //O(n)
3
4 foreach(dir in enum direction) //4 times
5     neighbour_pos = get_position_at_direction(card, dir) //O(1)
6     neighbour = retrieve_card_by_position(cards_set, neighbour_pos) //O(log(n))
7
8     if (neighbour exists AND neighbour can still be linked to a new card)
9         if (cards_matches_directions(card, neighbour, dir, opposed_dir)) //O(1)
10             ifnot (link_cards_at_dir(card, neighbour, dir, opposed_dir)) //O(1)
11                 remove_card_from_set(cards_set, card) //O(n)
12                 ret FAILURE
13         else
14             unlink_previously_linked_neighbours(card) //O(1)
15             remove_card_from_set(cards_set, card)
16             ret FAILURE
17
18 if (card doesn't have any neighbour)
19     remove_card_from_set(cards_set, card)
20     ret FAILURE
21
22 ret SUCCESS
```

Ainsi, la fonction `board__add_card()` utilisant ces dernières fonctions est de complexité :

$\theta(\log(n)) + \theta(n) + 4 \times (\theta(\log(n)) + \theta(1) + \theta(1)) + \theta(\log(n))$, c'est-à-dire $\theta(\mathbf{n})$.

Cette fonction est corecte car elle retourne soit *SUCCESS* (0), soit *!SUCCESS* en fonction des vérifications faites. Elle ajoute la carte au plateau dès le début puis l'enlève seulement dans les cas où il n'y a pas de correspondance, si le lien entre les cartes a échoué ou si la carte n'est liée à aucune autre.

4.2 Calcul des coups possibles

La procédure `client__populate_possible_moves_list()` est la responsable du calcul des coups possibles pour un joueur. Son prototype en C est le suivant :

```
1 void client__populate_possible_moves_list(struct client *c, struct set *
   possible_moves, enum card_id ci);
```

La fonction est écrite ci-dessous en pseudo-code avec la complexité des sous-fonctions. Vous pouvez trouver son code complet dans le fichier `src > client > client.c`

```
1 drawn_card = init_vard_from_id(card_id)
2
3 foreach (umpteenth in cards_set)
4     if (umpteenth can still be linked to a new card)
5         foreach (north_dir in enum direction)
6             drawn_card->direction = north_dir
7             foreach (dir in enum direction)
8                 if (cards_matches_directions(card, neighbour, dir, opposed_dir)
9                     AND umpteenth doesn't have neighbour at direction dir)
10                    drawn_card->pos = get_pos_at_dir(umpteenth, opposed_dir)
11
12                    if (adding_possible_for_others(board, drawn_card)) //O(n)
13                        onto = get_position_at_dir(umpteenth, opposed_dir)
14                        m = init_move(client_id, card_id, onto, north_dir)
15                        add_move_to_set(possible_moves, m) //O(n)
```

Elle construit la *set* de coups possibles avec les coups valides qu'elle trouve.

Cette procédure effectue une boucle sur le nombre de cartes sur le plateau qu'on note n . Sa complexité est de $\theta(n) \times 4 \times 4 \times (\theta(1) + \theta(n))$, c'est à dire $\theta(n^2)$.

Cette fonction est correcte car elle ne retourne rien et doit uniquement remplir la liste de coups possibles. Ce qu'elle fait via le pseudo-code `add_move_to_set()` après avoir vérifié qu'une carte peut être placée par rapport à ses potentiels voisins et sa direction nord.

5 Validation et qualité du développement

Cette section s'intéresse aux différentes méthodes utilisées dans notre projet afin de vérifier la qualité du code ainsi que la validité des coups. Elle se compose de trois parties : les tests unitaires, la vérification en commun du serveur et des clients et l'interface utilisateur.

5.1 Tests unitaires

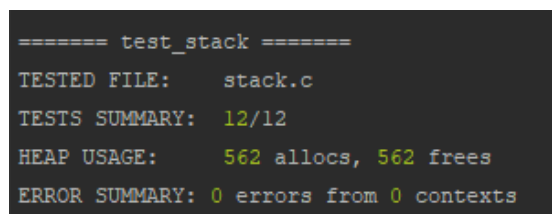
Afin de vérifier la validité de notre développement, nous avons réalisé des tests unitaires sur nos types abstraits de données (TADs) et l'ensemble des fonctions utilisés par le serveur et les clients à l'exception de la partie graphique et des fonctions englobantes (c'est à dire celle de `game.c` et `server.c`).

Pour les fichiers testés, nous avons créé pour chaque fonction du fichier au moins une et jusqu'à six fonctions de tests en fonction de la complexité⁴ de celle-ci. Nos tests unitaires affichent pour chaque module testé le nombre de tests en succès et le nombre total de tests lancés. Ce comportement correspond à la commande *make test*.

De plus, nous avons aussi inclus deux autres commandes qui agissent sur l'exécution des tests unitaires.

La commande *make vtest* exécute les tests précédents à travers l'analyseur de mémoire *Valgrind*⁵.

Pour chaque fichier testé, cette commande affiche un résumé à la fin de l'exécution de *Valgrind* contenant le nom du fichier testé, le nombre de tests en succès sur le nombre de tests en échec, le nombre d'allocations et libérations effectuées sur le tas (en vert si leur nombre correspond, en rouge si des *free* sont manquants) ainsi que le nombre d'erreurs mémoires (en vert si il n'y en a pas, en rouge sinon). La figure ci-dessous montre le type d'affichage que cette commande montre.



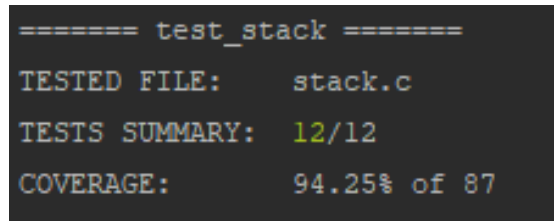
```
===== test_stack =====  
TESTED FILE:    stack.c  
TESTS SUMMARY:  12/12  
HEAP USAGE:     562 allocs, 562 frees  
ERROR SUMMARY:  0 errors from 0 contexts
```

FIGURE 3 – Affichage d'un test avec Valgrind

4. La complexité évoquée ici est celle en terme de lignes de codes et de sous-fonctions appelées et non la complexité algorithmique

5. *Valgrind* est un programme libre permettant de déboguer et d'identifier les *fuites mémoires* (espace mémoire alloué par le programme sur le tas qui n'est pas libéré d'ici à la fin de l'exécution du programme) et erreurs d'accès en mémoire (lectures/écritures invalides, variables non initialisées) des programmes sur Linux.

La commande `make ctest` exécute elle aussi les mêmes tests, mais cette fois-ci non pas avec *Valgrind* mais avec une couverture des tests qui utilise le programme *Gcov*⁶. Cela permet de savoir si les tests couvrent toutes les lignes des fonctions d'un fichier ou si des zones ne sont pas testées. Pour chaque fichier testé, cette commande affiche donc un résumé contenant le nom du fichier testé, le nombre de tests en succès sur le nombre de tests en échec, le pourcentage de couverture du fichier ($\frac{\text{nombre de lignes parcourues}}{\text{nombre de lignes utiles}}$) avec à côté le nombre de lignes utiles – une ligne utile n'étant ni une ligne vide ni une ligne de commentaire. La figure ci-dessous montre le type d'affichage que cette commande montre.



```
===== test_stack =====  
TESTED FILE:    stack.c  
TESTS SUMMARY:  12/12  
COVERAGE:      94.25% of 87
```

FIGURE 4 – Affichage d'un test avec Valgrind

5.2 Vérification par mise en commun des clients

Cette initiative proposée par notre encadrant de projet nous permet lors des séances de projet de vérifier l'inter-compatibilité de nos clients et nos serveurs parmi tous les groupes de projet. Cela permet d'identifier des erreurs de validité des coups ou de déterminer si les clients et du serveur sont implémentés de façon générique, et éviter de créer un client en supposant que notre serveur va réagir d'une certaine façon. En effet, la façon d'utiliser les structures de données sont propres à notre groupe, et certaines fonctions sont communes entre notre client et notre serveur, le serveur qui vérifie les coups selon les mêmes principes par exemple, ce qui peut nous empêcher de détecter des erreurs.

Le premier cas évoqué est indispensable, car les clients et serveurs implémentent généralement de la même façon la validité des coups donc même si l'un n'est pas valide, il se peut que le coup soit quand même accepté. Cela s'est d'ailleurs produit dans notre cas, jusqu'à ce qu'on teste notre serveur avec une carte valide d'un autre groupe.

6. *Gcov* est un programme libre analysant la couverture du code source lors de l'exécution d'un programme. C'est-à-dire le profilage du nombre de fois ou une ligne de code est parcourue

5.3 Interface utilisateur

Nous avons réalisé une interface utilisateur dans les 2 avant-dernières semaines, afin de pouvoir plus facilement vérifier la validité des coups, ce qui était peu évident avec les affichages de debug en console. Cela nous a permis d'identifier un dernier cas de mauvaise validité d'une carte dans le cas où toutes les directions correspon-daient sauf la dernière direction vérifiée.

Cette interface utilisateur inclut un écran de menu (image ci-dessous à gauche) composé d'un bouton *play*, d'un bouton *quit* et affichant le nombre de clients prêt à jouer, ainsi qu'un écran de jeu affichant la pile des cartes à piocher, le plateau du jeu, ainsi qu'un bouton *pause*. Il est possible de retourner au menu avec la touche *échap*. Le taille du plateau de jeu s'adapte dès que le plateau atteint les bords de l'écran afin que l'ensemble des cartes soit toujours visible.

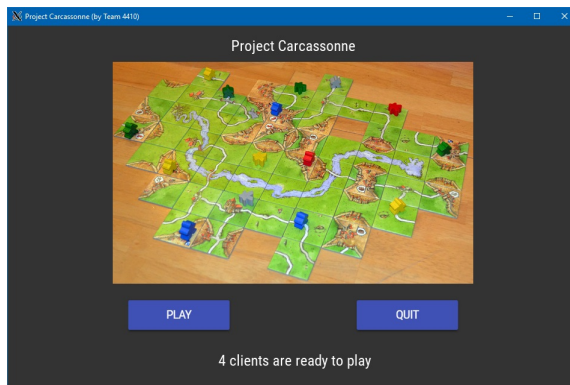


FIGURE 5 – Écran de menu

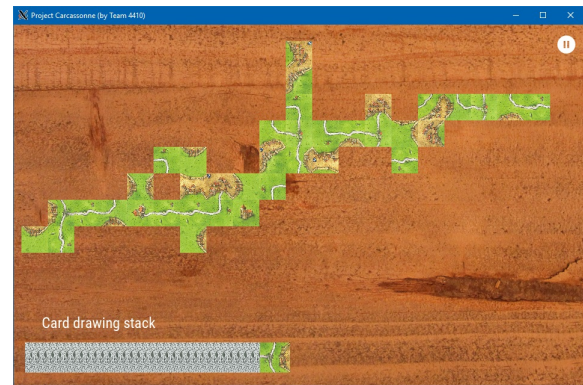


FIGURE 6 – Écran de jeu

6 Problèmes rencontrés

Dans cette section, nous présentons les problèmes techniques principaux que nous avons rencontrés au cours de la réalisation du projet.

6.1 Vérification de correspondance entre 2 cartes

La conception de la solution algorithmique permettant de vérifier si 2 cartes correspondent a été le principal problème rencontré lors du développement de ce projet. Elle a également été la fonctionnalité la plus longue à réaliser pour être 100% fonctionnelle.

En effet, nous avons débuté cette fonctionnalité fin avril et n'avons pu corriger le dernier cas particulier que grâce à l'interface graphique, donc vers la fin du projet. Ce problème additionné aux nombreuses séances manquantes a, de ce fait, retardé de quelques semaines le début de travail sur le placement des *meeples* qui n'a pu commencer que le 11 mai.

Nous pensons que ces problèmes sont dûs, en premier lieu, au manque de réflexion sur le papier avant de l'implémenter en C, contrairement au placement des *meeples* et à la gestion des scores que nous avons détaillée ensemble sur le papier mais n'avons pas eu le temps d'implémenter. En deuxième lieu, au fait que nos tests unitaires ne géraient pas tous les cas et donc passaient. De plus, vérifier le placement des cartes à partir de fonctions de debug n'était pas évident en particulier lorsque les cartes avaient une direction différente de *NORTH*, qui plus est sachant que chaque groupe donnait une interprétation différente du sens dans lequel la carte devait tourner suivant sa direction *WEST* (cf. 6.4).

6.2 Utilisation de fichiers JPEG pour l'affichage des cartes avec SDL2

Ce problème, contrairement au précédent n'a duré qu'une semaine mais les nombreuses heures passées à tenter de le déboguer auraient pu être évitées car il s'agissait d'un problème externe.

En effet, ce problème impactait l'affichage des textures de cartes dont les fichiers au format *JPEG* ont été récupérés sur le répertoire de M. Renault.

Celles-ci n'étaient pas correctement chargées par *SDL2Image*, c'est à dire qu'elles étaient affichées dans une mauvaise direction (même avec une rotation à 0) et malgré que le fichier *JPEG* ait été édité afin qu'elle pointe par défaut sur le même *NORTH* que décrit dans le fichier *interface.h*.

Par exemple, la carte de départ *CARD_ROAD_STRAIGHT_CITY* était affichée par défaut comme si elle avait la direction *SOUTH*, aucune autre carte ne suivait la même logique. Nous avons d'abord pensé que le problème venait de notre implémentation ou même de la fonction *SDL_RenderCopyEx()*, mais nous avons à un moment eu l'idée de changer le format de fichier en remarquant que nos autres images (menu, boutons,...) qui, elles, étaient au format *PNG* se chargeaient correctement et pouvaient être tournées dans le sens voulu.

Nous avons converti la première carte au format *PNG*, ce qui a corrigé le problème,

et nous avons donc décidé de faire de même pour les 22 autres.

Au final, nous ne savons pas s'il y a effectivement un bug dans la version actuelle de la *SDL2Image* ou si les images *JPEG* étaient en quelque sorte "corrompues".

6.3 Ajout de copies dans les TADs

Sur le modèle d'une de nos implémentations du TAD *set* en PG116, nous avons opté pour une allocation dynamique des données avec stockage de copies. Cette dernière caractéristique a entraîné une difficulté supplémentaire à gérer dans le programme (désallocation d'objets une fois insérés dans le TAD, et désallocation d'objets dont la copie est récupérée depuis le TAD) et une non-compréhension de certains bugs comme le lien entre les cartes qui ne se faisait pas sur les bonnes cartes lors des tests unitaires.

6.4 Interprétation différente du sujet selon les groupes

Cette dernière difficulté est entrée en jeu lorsque nous avons commencé à échanger nos serveurs et clients car ceux-ci devaient être inter-compatibles, au moins pour les groupes de la même salle de projet. Quelques interprétations de la façon dont certaines situations devaient être gérées par le serveur ou même la façon dont une rotation devait être gérée ont entraîné des débats intenses qui ont même nécessité un arbitrage par le concepteur du projet.

Les éléments suivants ont été évoqués :

- La position de la carte de départ en $\{ 0, 0 \}$
- Le sens de positionnement vertical des cartes : +1 pour *NORTH*, -1 pour *SOUTH*
- Le fonctionnement de la fonction *get_player_name()* qui pour certains groupes devait demander à l'utilisateur d'entrer le nom du joueur, et qui pour d'autres ressemblait plus à un simple *getter* permettant à n'importe quel moment de récupérer le nom du joueur sans aucune interaction avec l'utilisateur. La deuxième solution a été choisie de manière bilatérale.
- Le fonctionnement du serveur lorsque un client effectuait un mauvais coup. Il a été décidé que le client était exclu du jeu mais que son coup devait être gardé en mémoire tout comme ses *meeples* devaient rester sur le plateau.
- Et enfin l'interprétation de comment une direction de carte devait être appliquée. Il y avait pour ce problème deux groupes opposés. L'un considérait que mettre la direction à *WEST* signifiait considérer l'ouest comme devenant la direction nord, ce qui semblait plus naturel dans la réalisation (décalage dans le même ordre qu'*enum direction* [+1], et sens de rotation identique à celui de la *SDL2* [90°]. Le second considérait que le nord se trouvait désormais à l'ouest. C'est cette dernière solution qui a été choisie.

7 Pistes d'amélioration

Cette section traite des différentes idées d'amélioration de notre projet dans son état actuel, suivant les fonctionnalités qu'il reste à faire ainsi que le regard que l'on peut porter sur des problèmes rencontrés et détaillés à la section précédente.

7.1 Fonctionnalités restantes : placement des *meeples*

Ainsi que présenté dans la section 3.3.4, la gestion des *meeples* était en cours de développement. Il s'agit d'une partie importante du jeu Carcassonne tant du point de vue du serveur que du point de vue des clients. Le serveur est censé autoriser ou non le placement du *meeples* sur la carte jouée par un joueur, et ceci est conditionné au placement des autres *meeples* déjà en jeu. Le client peut choisir l'emplacement sur la carte où poser son *meeples*, selon les *meeples* déjà posés. Il est donc question d'un parcours du plateau de jeu afin de savoir délimiter des zones communes. Un critère qui a été délicat à définir est l'appartenance de deux *area_type* (qui est la plus petite unité pour décrire les cartes) à la même zone. D'après notre modèle fourni en Figure 2, il arrive que deux cartes partagent les mêmes types de zones aux mêmes endroits, ou encore qu'au sein d'une même carte la même zone se retrouve sur deux *area_type* non adjacentes, ce qui engendre de nombreuses difficultés. La piste que nous étions en train de creuser consistait à utiliser le TAD *set* déjà existant pour stocker les différentes zones du plateau, plutôt que de parcourir un graphe, en construisant à l'avance les zones propres à chaque carte. Puis quand une carte est placée sur le plateau, il suffit de fusionner les différentes zones aux zones déjà présentes dans le plateau et stockées dans un *set*.

7.2 Fonctionnalités restantes : comptage des points

Le comptage des points devait être géré par la structure de données *zone*. En effet une structure *zone* contient un champ *score* qui sera augmenté au fur et à mesure que la zone est complétée. Lorsqu'une zone est fermée, l'attribution des points à chacun des joueurs se fait grâce aux *meeples* se trouvant sur la zone qui contiennent les *id* des joueurs auxquels ils appartiennent.

7.3 Fonctionnalités restantes : stratégies de jeu

Nous aurions souhaité développer des stratégies de jeu pour chaque client, afin que ceux-ci jouent différemment selon la forme actuelle du plateau et les *meeples* déjà placés.

Nous avons déjà mis en place une structure permettant d'implémenter une stratégie dans nos clients. En effet, lorsque la fonction *play()* des clients est appelée, nous avons séparé la mise à jour du plateau, la récupération des coups possibles ainsi que la partie de choix du coups afin qu'une seule fonction soit responsable de la stratégie. Celle-ci est définie dans notre fichier *client.h* et doit être implémentée dans chaque client puis appelée dans *play()*. C'est la seule fonction du client qui a

besoin d'être modifiée pour adapter la stratégie. Elle suit le prototype suivant :

```
1 struct move client__choose_move_strategy(struct micro_board *board, unsigned
    int nb_meeples, struct set *possible_moves);
```

Elle prend en paramètre le plateau de jeu dans son état actuel, le nombre de *meeples* pouvant encore être placés ainsi que la liste des coups possibles.

Dû au fait que nous n'avons pas eu le temps d'implémenter le placement des *meeples* et la gestion des scores, nous n'avons écrit que des clients avec des stratégies rudimentaires :

- *aléatoire valide* : choisit un coup aléatoire parmi les coups possibles.
- *toujours en bas à gauche* : choisit toujours le coup plaçant la carte le plus à gauche puis le plus en bas, car la première qu'il peut jouer en parcourant le *set* trié.
- *toujours en haut à droite* : choisit toujours le coup plaçant la carte le plus à droite puis le plus en haut, en jouant la première carte jouable depuis la fin du *set*.

7.4 Conception des TADs sans copie

Comme précédemment évoqué dans la section 6.3, le stockage de copies dans nos TADs n'était pas une très bonne idée compte tenu des problèmes rencontrés. En effet, nous avons très souvent oublié que l'on manipulait des copies, par habitude, pour au moins l'un d'entre nous de manipuler les TADs de la *Standard Template Library* de C++ (qui stocke directement les pointeurs des objets ajoutés), ou simplement par oubli.

En faisant le bilan du projet, nous pensons donc qu'il serait préférable, pour éviter des problèmes, de n'utiliser que des TADs sans copie si nous avons l'occasion d'en recréer à l'avenir.

8 Conclusion

Ce projet nous a été bénéfique en terme de travail collectif et de répartition des tâches.

Nous avons pu découvrir le chargement de bibliothèques dynamiques grâce au développement de l'interface d'échange entre le client et les serveurs ainsi que le développement d'une interface graphique.

Nous avons également eu l'occasion de mettre en pratique l'outil CMake pour la compilation du projet et de renforcer notre compréhension de la gestion de la mémoire ainsi que la détection de fuites mémoire grâce à Valgrind.

A l'échéance de ce projet, nous avons réussi à implémenter un serveur échangeant avec des clients renvoyant des coups valides. Cependant, nous aurions souhaité pouvoir gérer le placement de *meebles* ainsi que le comptage des points.